SUBMITTED TO: SOFTWARE - PRACTICE AND EXPERIENCE

Fast Incremental Compiler Transformations For Multiple Instruction Retry

Shyh-Kwei Chen, Neal J. Alewine*, W. Kent Fuchs, and Wen-Mei W. Hwu

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main Street
Urbana, IL 61801

Correspondent: Shyh-Kwei Chen
Tel: (217) 244-7180
FAX: (217) 244-5686
Email: skchen@crhc.uiuc.edu

Abstract

Previous work on compiler-assisted multiple instruction retry has utilized a series of compiler transformations, *loop protection*, *node splitting*, and *loop expansion*, to eliminate anti-dependencies of length $\leq N$ in the *pseudo register*, *machine register*, and *the post-pass resolver* phases of compilation. The results have provided a means of rapidly recovering from transient processor failures by rolling back N instructions. This paper presents techniques for improving compilation and run-time performance in compiler-assisted multiple instruction retry. Incremental updating enhances compilation time when new instructions are added to the program. Post-pass code rescheduling and spill register reassignment algorithms improve the run-time performance and decrease the code growth across the application programs studied. Branch hazards are also shown to be resolvable by simple modifications to the incremental updating schemes during the pseudo register phase and to the spill register reassignment algorithm during the post-pass phase.

Index terms: rollback recovery, fault-tolerant computing, instruction retry.

---

*IBM, Boca Raton Fl.

This research was supported in part by the National Aeronautics and Space Administration (NASA) under grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), and in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under Contract N00014-91-J-1283.

1 Introduction

Software based checkpointing provides for rollback recovery when transient system faults occur. In such schemes, a checkpoint of the system state is captured and recorded at regular intervals [2, 3, 4], or predetermined positions in the application program [5]. In the event of a fault, the system can be rolled back to one of the previously recorded checkpoints, returning the system to a consistent state [6]. Software checkpointing can accommodate long error detection latencies at the cost of potentially long recovery time.

In contrast to full software checkpointing, multiple instruction retry schemes aid in rollback of just a few instructions, requiring shorter error detection latencies while resulting in less lost work during recovery. Instruction retry schemes have traditionally been implemented in hardware, both in full checkpointing [7, 8], and in incremental checkpointing (sliding window) [9, 10] formats.

Recently, a compiler-assisted multiple instruction retry scheme has been developed in which compiler-driven data flow manipulation is used to resolve data hazards associated with rollback recovery [1]. Anti-dependencies of length $\leq N$ are eliminated using a series of compiler transformations. A combined compiler-hardware scheme [11] has also been developed which can remove one type of hazard while allowing the compiler-driven transformations to resolve the remaining hazards.

This paper provides compilation and run-time performance enhancements that have been implemented for compiler-assisted multiple-instruction retry. The techniques described include incremental updating, post-pass code rescheduling, spill register reassignment and branch hazard resolution. Implementation and performance benefits of the schemes are evaluated on a set of twelve programs which are cross-compiled on a SPARC server 490 and executed on a DEC station 3100.

2 Error Model and Hazard Types

Targeted processor errors are described as follows [11]. Error detection latency is $\leq N$ instructions. Units external to the CPU, such as memory and I/O, have their own rollback capability (e.g., delayed write buffers of depth N and appropriate bypass logic). The program counter contents at each instruction are preserved by an external recording device or by shadow registers [9]. A

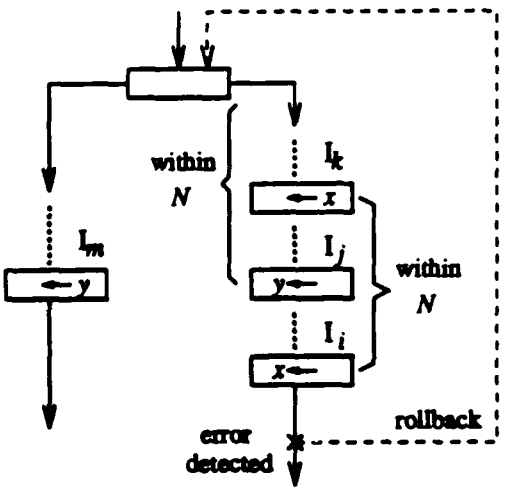


Figure 1: On-path and Branch Hazards.

restartable CPU state can be restored by loading the correct contents of the register file and the program counter.

Given the above assumptions, a permissible error is one which does not result in a path inconsistent with the control flow graph (CFG) of the target application program provided that the register file contents do not spontaneously change and data is not written to an incorrect register location. Errors targeted for recovery via multiple instruction retry are summarized as follows: 1) CPU errors such as those caused by a faulty ALU; 2) incorrect values read from memory, the register file, or external functional units such as the floating point unit; 3) correct/incorrect operands read from incorrect locations within the I/O, memory, or register file; and 4) incorrect branch decisions resulting from errors 1 through 3.

The code can be represented as a CFG, $G(V, E)$, where V is the set of nodes denoting instructions and E the set of edges denoting flow information. If there is a direct control flow from instructions I_i to I_j, where $I_i \in V$ and $I_j \in V$, then there is an edge $(I_i, I_j) \in E$.

Within the general error model above, data hazards resulting from instruction retry are of two types [11]. On-path hazards are those encountered when the instruction path after rollback is the same as the initial instruction path and branch hazards are those encountered when the instruction path after rollback is different from the initial instruction path. On-path hazards can also be described as anti-dependencies of length $\leq N$ in $G(V, E)$ [12]. As shown in Figure 1, register x of instruction I_i represents an on-path hazard and register y of instruction I_j represents a branch

Table 1: Schemes implemented

	Pseudo register	machine register	Nop insertion
Scheme L	on-path	on-path	on-path
Scheme A	on-path + branch[*]	on-path + branch	on-path + branch
Scheme 0	on-path[i]	on-path	on-path[cr]
Scheme 1	on-path[i]	on-path	on-path + branch[cr]
Scheme 2	on-path + branch[i]	on-path	on-path + branch[cr]
Scheme 3	on-path + branch[i]	on-path + branch	on-path + branch[cr]

hazard.

3 Overview of Schemes Implemented

In order to compare compile time and run time efficiency, we have implemented several schemes for each of the phases, as shown in Table 1. Data hazards are resolved at three different phases. The *pseudo register* phase employs loop protection, node splitting, and loop expansion to resolve hazards at the pseudo register level. The *machine register* phase performs register allocation to resolve machine register hazards, and the *nop insertion* phase resolves the remaining hazards by inserting any required nops at the assembly code level.

Scheme L [1] resolves on-path hazards only, and Scheme A [11] resolves both on-path and branch hazards at all three phases. Scheme A does not resolve all pseudo register branch hazards due to loop expansion, as marked "[*]". The dominant fraction of compile time in previous Schemes L and A is devoted to resolving pseudo register hazards. Both schemes implement a simple pseudo register phase. Loop protection, node splitting, and loop expansion may insert new instructions which can change the loop structure, dataflow information, and may therefore create new hazards. Since the data structure updating is not incrementally maintained, both previous schemes repeat each stage for all loops until there are no new instructions to insert.

In addition to previous schemes L and A, we implemented four alternative schemes that exploit incremental compilation techniques. Scheme 0 uses incremental updating in the pseudo register phase for resolving on-path hazards. Compilation time has been enhanced with respect to Scheme L. Scheme 0 also employs post-pass code rescheduling and spill register reassignment algorithms to enhance the run-time performance and decrease the code growth across the application programs

studied. The marker "[i]" denotes incremental updating, while "[cr]" denotes code rescheduling. Modifications to the post-pass algorithms can resolve both types of hazards during the nop insertion phase (Schemes 1, 2, and 3). We also show that a slightly modified incremental updating scheme can resolve branch hazards as well in the pseudo register phase (Schemes 2 and 3), though experimental results favor Scheme 1 in code run-time, code growth and compilation speed.

4 Review of the Pseudo Register Phase in Scheme L

The following notation is for on-path hazards, while those for branch hazards can be similarly defined. An instruction I_i is a *hazard instruction* if I_i defines a register x, another instruction I_j uses x, and there is a directed path of length less than or equal to N from I_j to I_i. Register x is called a *hazard register* or a *hazard* that causes data inconsistency. An instruction I_j will be split due to hazard register x if $x \in live\_in(I_j)$ and there is more than one definition of x that can reach I_j. Loop expansion, combined with renaming, is used to increase the anti-dependency distance to exceed N within loops. To prevent some loop headers from being split, and to allow the targeted hazards to be renamed freely after loop expansion, save and restore nodes are inserted around loop headers, tails, and trailer nodes. A loop can be protected either from outside or from inside. The former executes save and restore instructions exactly once, while the latter executes save and restore instructions for every loop iteration. It is obvious that a loop protected from outside executes fewer instructions when the loop is executed at least twice. The saved registers within the loop are renamed to corresponding new registers. The following conditions are used to determine if a loop L should be protected for register r:

C1. r is a hazard register which is live after the extended loop $\tilde{L}$ for register r.

C2. L's header will be split due to its hazard register r.

C3. L's header will be split due to out of loop hazard register r.

The extended loop $\tilde{L}$ for register r consists of all nodes in L and all nodes I_i satisfying the following rules: 1) $r \in$ live_in(I_i), 2) I_i has only one successor, 3) I_i has only one predecessor I_j, and 4) I_j is in $\tilde{L}$. If C1 or C2 is true, L is protected from inside. If C3 is true, L is protected from outside. L is not protected for r if none of the three conditions hold. C3 prevents L's header from being split,

while C1 and C2 confines r's live range to within each iteration of L, so that after loop expansion, r can be renamed correctly within each new loop copy. C1 is for $\tilde{L}$ instead of L since L may not have to be protected if all nodes in which r is live after L are in $\tilde{L} - L$.

To limit the compilation time and code growth, a threshold is set to 800 instructions so that the procedure aborts normal evaluation of the loops, and simply inserts enough nops to resolve the remaining hazards should the code size exceed the threshold. Since we will illustrate our ideas using real program segments, we list the benchmarks with code sizes and descriptions as follows : QUEEN(148), 8-queen program; QSORT(261), recursive quick sort algorithm; PUZZLE(877), a game; WC(181), CMP(251), GREP(926), COMPRESS(1828), UNIX utilities; EQN(6251), mathematics typesetting program; LEX(6873), lexical analyzer; YACC(8099), parser generator; CCCP(8775), preprocessor for gnu C compiler; and TBL(9191), table formatter.

5 Performance Enhancement Techniques

In this section, we discuss techniques that can enhance code run time and reduce code growth at the pseudo register phase. Loop L is protected from inside for register r if condition C1 or condition C2 is true. However, L may have a special property which allows the save/restore nodes for r to be moved out of the loop. This can save code run time since the save/restore nodes are only executed once for every iteration of L. For register r, if any header to tail path within loop L has at least one instructions defining r, then there exists suitable renamings along certain cut line on these paths after L has been expanded. We introduce the notions of the *cut register set* and the *cut node set* as follows:

Definition (1) HR_i and HN_i are the set of hazard registers and hazard nodes respectively within loop L_i. (2) CR_i is the cut register set of loop L_i. Register $r \in CR_i$, *iff* any directed path leading from L_i's header to some tail has one or more instructions defining r. (3) CHR_i is the cut hazard register set of loop L_i. $r \in CHR_i$, if $r \in CR_i$ and $r \in HR_i$. (4) $CNL_i(r)$ is the cut node set of loop L_i for register r. For any L_i's loop node α, $\alpha \in CNL_i(r)$, *iff* $r \in CR_i$, α defines r and there exists at least a directed path from α to at least one of L_i's tails that no node, except α, on this path defines r. (5) Let $d_L(I_i, I_j)$ denote the minimum number of edges on any path within loop L from I_i to I_j, and D_L denote the minimum number of edges from L's loop header to any of L's tail.

(I_α, I_β) is a *hazard pair* within loop L on register r if I_α uses r, I_β defines r, and $d_L(I_\alpha, I_\beta) \leq N$.

5.1 Loop Protection

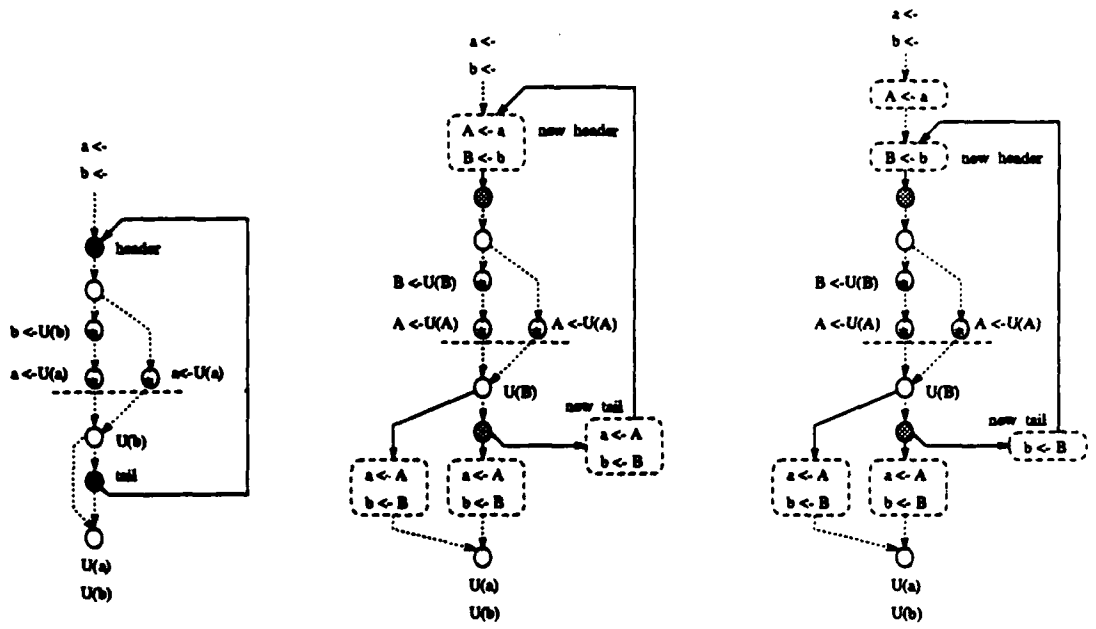
Consider that loop L_i needs to be protected from inside for r. If $r \in HR_i$, and $r \notin CR_i$, then the save node $r' \leftarrow r$ is in $CNL_i(r')$, and all the restore nodes $r \leftarrow r'$ are in $CNL_i(r)$. Note that r' is a new register and all references of r within L_i are renamed to r'. If $r \in CHR_i$, then loop L_i can be renamed correctly after being protected from outside for r and expanded a sufficient number of times. Similarly, if $r \in HR_i$, $r \notin CR_i$, and r is dead at the header of L_i, then L_i can be renamed correctly after being protected from outside for r with the removal of the save node and expanded a sufficient number of times.

Example Figure 2(a) is a program segment for the first function of QSORT. A "*" sign within a circle denotes a hazard node, and $U(x)$ represents using register x. Dotted lines denote that there may be some nodes in between as long as they do not redefine the targeted registers. Solid lines denote no instructions in between, and dashed lines denote cut lines. If we apply the original loop protection algorithm, the loop is protected from inside for both registers, as shown in Figure 2(b). Every loop iteration executes four additional instructions. We can move the save/restore nodes for register a out of the loop since they belong to the cut hazard register set of the loop. Figure 2(c) illustrates such transformation, and now every loop iteration executes two additional instructions.

5.2 Node Splitting

For any given node I, let L be the number of incoming edges, and M be the number of original reaching definitions that can reach I, among which K definitions are hazards. We have implemented a scheme in which the number of copies, S, for node I after splitting satisfies the following criterion: 1) $S = K$, if $M = K$; or 2) $S = K + 1$, if $M > K$.

This can be done by using a stamp heap data structure [13], so that if a hazard node I is split into $I_1, I_2, \ldots, I_S$, then the stamp field of I_i, $i = 2, 3, \ldots, S$, points to I_0. The hazard nodes in the same heap will be assigned to the same new destination register if renaming is required. Therefore a node I with hazard x in its *live_in* set will be split if 1) there are more than one reaching definitions of x, and 2) all reaching definitions of x do not belong to the same stamp heap, assuming that all non-hazard nodes defining x belong to the same stamp heap.



(a) QSORT segment (b) Inside protect for register a (c) Outside protect for register a

Figure 2: Cut registers and cut nodes for loop protection

5.3 Loop Processing Order

Node splitting transforms all the hazards within the current loop across its backedges, while loop expansion resolves all such hazards. In this manner, when we process a given loop, there is no data hazard across the backedges of its inner loops. Therefore it is natural to process the loops from inside out so that the levels of data hazards can be successively reduced until all of them occur at the root level. The hazards at the root level then can be resolved by node splitting and renaming.

In addition to the inner loop first rule, we have to enforce the sequential order rule (top-down) to smoothly check condition $C3$ in Section 1 for parent hazard registers and to further eliminate extra save/restore nodes. Suppose that h_1 and h_2 are loop headers of L_1 and L_2 respectively, and there is a directed path from h_1 to h_2 without visiting any backedge. According to the loop nesting property, we have two cases : 1) L_2 is an inner loop of L_1; or 2) L_2 is sequentially after L_1, as shown in Figure 3(a), and 3(b). For case 1, L_2 is processed before L_1, while for case 2, L_1 is processed before L_2. If, without visiting any backedge, there is no directed path from h_1 to h_2 or from h_2 to h_1, then either L_1 or L_2 can be processed first, as shown in Figure 3(c). Figure 3(d) illustrates the inner loop first rule, that new hazards due to loop protection will be propagated to the outer

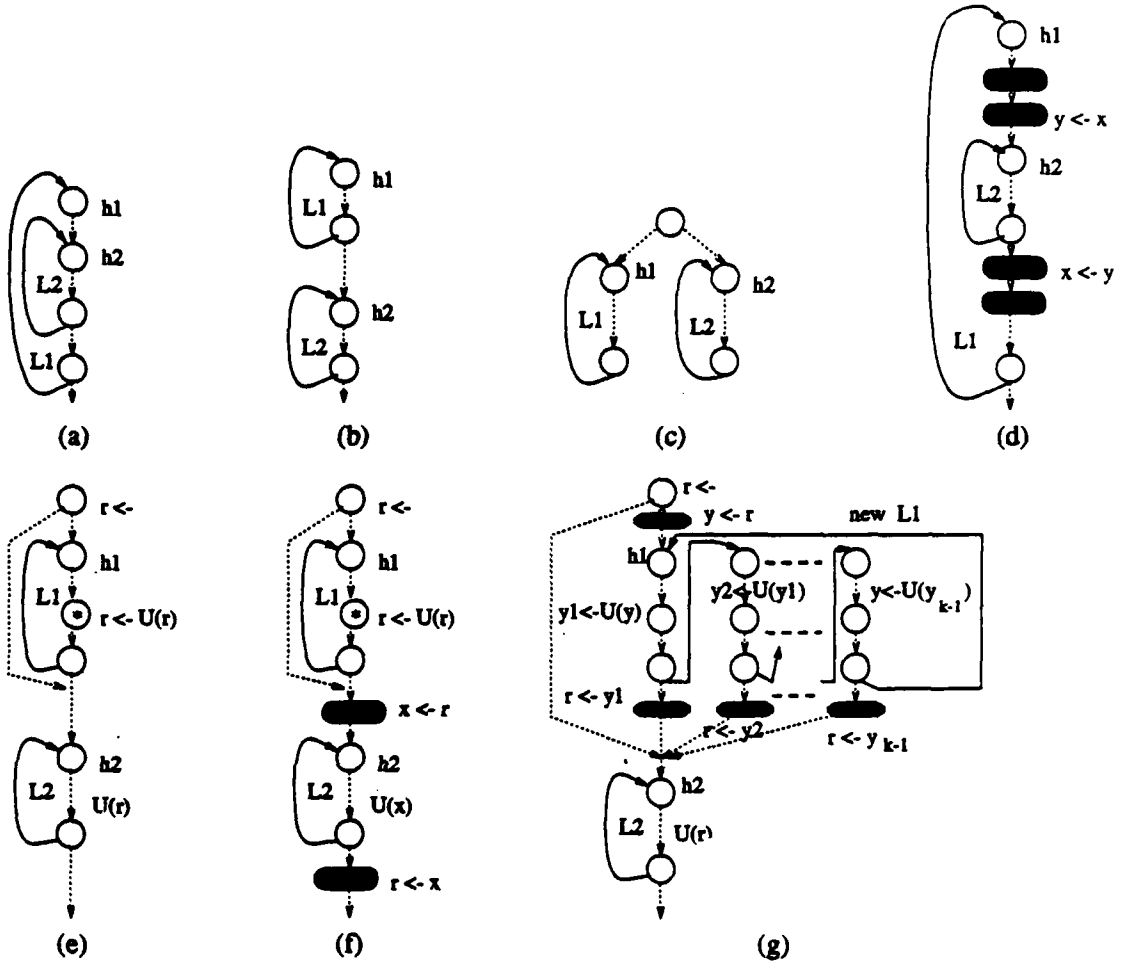


Figure 3: Loop processing order

loop. Figure 3(e) is a real program segment (the first function of CMP) and can illustrate the sequential order rule. Suppose L_2 is processed first. Without enforcing the sequential order rule, L_2 may need to be protected from outside for register r, as shown in Figure 3(f). However, such protection is redundant if we process L_1 first and remove its hazards that might affect L_2, as shown in Figure 3(g).

Breadth First Search (BFS) is used to determine the processing order of nodes within loops or nodes of the entire program. The starting nodes may be the headers of loops or the root of the program. For some procedures, we have to modify the BFS algorithm by enforcing the following rules : 1) a node can be processed if and only if all of its parents have been processed, MBFS; 2) bypass inner loops, BBFS; 3) reverse the direction of searching, RBFS.

5.4 Loop Expansion

Our formula for the number of copies of L needed to resolve all on-path hazards within L is different from the previous work [1] due to the cut-register-set. To simplify the analysis, we assume that loop L has a header I_h, and a single tail I_t. It can be easily extended to loops with multiple tails. Let $D_L = d(I_h, I_t)$. Assume that (I_i, I_j) is a hazard pair within loop L for register x. The new formula includes the following cases: Case 1. The backedge (I_t, I_h) is not counted in $d_L(I_i, I_j)$; Case 2. The backedge (I_t, I_h) is counted in $d_L(I_i, I_j)$, and within L there exists a directed path that does not include (I_t, I_h) from I_j to I_i; and Case 3. The backedge (I_t, I_h) is counted in $d_L(I_i, I_j)$, and within L not considering (I_t, I_h), there is no directed path from I_j to I_i.

Suppose it takes K_1, K_2 and K_3 copies to resolve the hazard pair (I_i, I_j) for each case respectively. We have $K_2 = \left\lfloor \frac{N - d(I_i, I_t) - d(I_h, I_j) - 1}{D_L + 1} \right\rfloor + 2$, and

$$K_1 = K_3 = \begin{cases} 2 & \text{, if } d(I_i, I_t) + d(I_h, I_j) + 1 > N \\ \left\lfloor \frac{N - d(I_i, I_t) - d(I_h, I_j) - 1}{D_L + 1} \right\rfloor + 3 & \text{, otherwise.} \end{cases}$$

The number of copies of L needed to resolve all hazards within L is the maximum of all such K's. Note that the number of expansions is at least 2.

5.5 Self-Anti-Dependent Instructions

An instruction I is *self-anti-dependent* if I uses the definition that it defines. For example, $x \leftarrow x + a$ is a self-anti-dependent instruction that defines and uses pseudo register x. This type of anti-dependency can be resolved by splitting I into two instructions : ($I_1 : y \leftarrow x + a$, $I_2 : x \leftarrow y$), and then inserting N nops between them [1, 11]. However, using renaming with the aid of node splitting and loop protection, we can rename the definition of x to a new pseudo register without inducing one new instruction.

6 The Incremental Updating Scheme

6.1 For On-Path Hazards – Scheme 0

Figure 4 shows the flowchart of the incremental scheme for on-path hazards during the pseudo register phase. Three subroutines *loop-protection*, *node-splitting*, and *replicate-loop*, marked by "*",

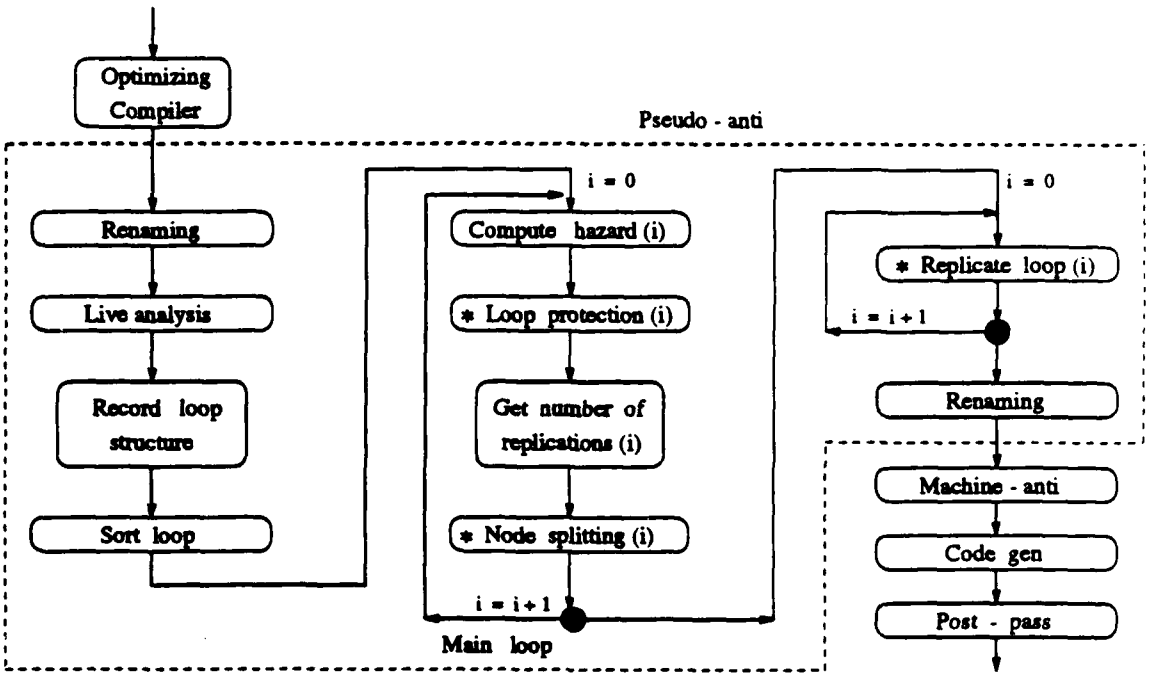
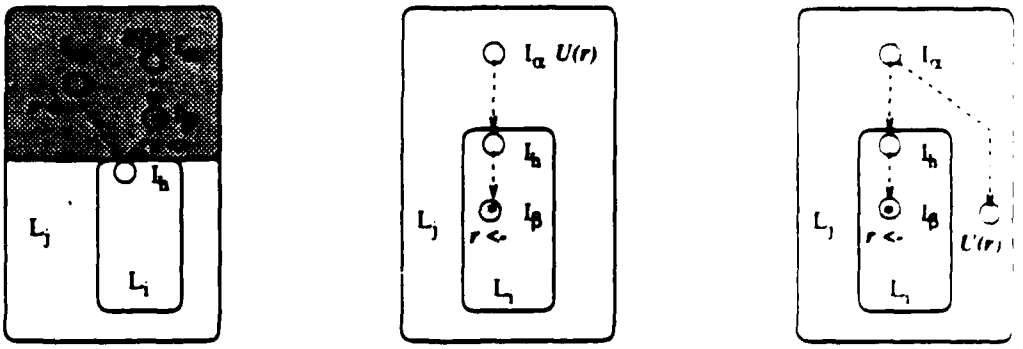


Figure 4: Incremental updating for on-path hazards

may insert extra nodes around or within loops. Information associated with each node, including register live range, stamp heap and loop structure, is updated locally whenever a node is inserted.

To determine if the header I_h of an inner loop L_i will be split due to some parent loop hazard register r, we have to check the nodes outside of L_i (condition C3). We confine the search for such hazard pairs to across L_i, or for hazard nodes to within L_i's immediate parent loop, but not within L_i. Assume that L_j is L_i's immediate parent loop or the entire program (root level) if L_i has no parent loop, and (I_α, I_β) is a hazard pair for register r. The two cases in which we consider protecting L_i from outside for r are shown in Figure 5(a) and (b). In Figure 5(a), since the r in I_β will be renamed, we only need to check if there is any other definition of r, I_ω, that can reach I_h, and is not in the same stamp heap as I_β. The search for I_β is restricted to the shaded area, denoting the definitions within L_j that can reach I_h without going through backedges, but I_ω can be nodes in the upper levels that can reach I_h. The hazard in Figure 5(b) can also be resolved by expanding L_i a sufficient number of times and renaming registers within L_i. For simplicity, we protect L_i from outside for register r instead, so that the hazard is automatically resolved. For other cases, the paths that cause hazards either belong to the current loop, which is detectable



(a) Hazard splitting the loop header. (b) Across loop on-path hazard. (c) Across loop branch hazard

Figure 5: The confinement of search nodes outside the loop

when we process the current loop, or belong to outer loops, which will also be found when we handle the outer loops subsequently.

6.1.1 Preparation

Subroutines *renaming*, *live-analysis*, *record-loop-structure*, and *sort-loop* are executed only once. The incremental scheme does not perform global DU-chain and global reaching definition analysis as Scheme L does, but rather performs a global live range analysis. After the preparation, loop information and dataflow information *live_in* and *live_out* are maintained and updated locally throughout the computation. Loop processing order is determined by a *sort-loop* subroutine, which evaluates loops in a top-down order in addition to the inner loop first rule.

6.1.2 Main Loop

The primary functions of loop expansion include : 1) compute the number of copies needed to resolve all on-path hazards within loops; 2) replicate loops; and 3) rename all registers within loops. When the *compute-hazard* subroutine bypasses inner loop hazards, functions 2 and 3 can be moved out of the main loop without affecting the correctness of the new scheme. Therefore, the main loop consists of *compute-hazard* , *loop-protection* , *get-number-of-replications* , and *node-splitting* subroutines, and each iteration evaluates one loop, as shown in Figure 4. This strategy is efficient since the actual code growth (function 2), is outside the main loop.

Subroutine *compute-hazard* computes HR_i, and HN_i, bypassing inner loop hazards. It traverses nodes within L_i from the loop header in a BFS order. If node I defines x, it performs an RBFS

traversal from node I up to distance N, but the search never leaves L_i. $x \in HR_i$ and $I \in HN_i$ *iff* there is a use of x within distance N. Subroutine *loop-protection* protects loop L_i according to criteria C1, C2, and C3.

Subroutine *get-number-of-replications* performs a BFS traversal to compute $D_{h,\beta}$ and an RBFS traversal to compute $D_{\alpha,t}$ for all nodes I_α, I_β in L_i. It then computes $\left\lfloor \frac{N-d(I_\alpha,I_t)-d(I_h,I_\beta)-1}{D_{L_i}+1} \right\rfloor + k$ according to the new formula, for every hazard pair (I_α, I_β) in L_i. The maximum of all such values is the number of replications needed for L_i to resolve its hazards.

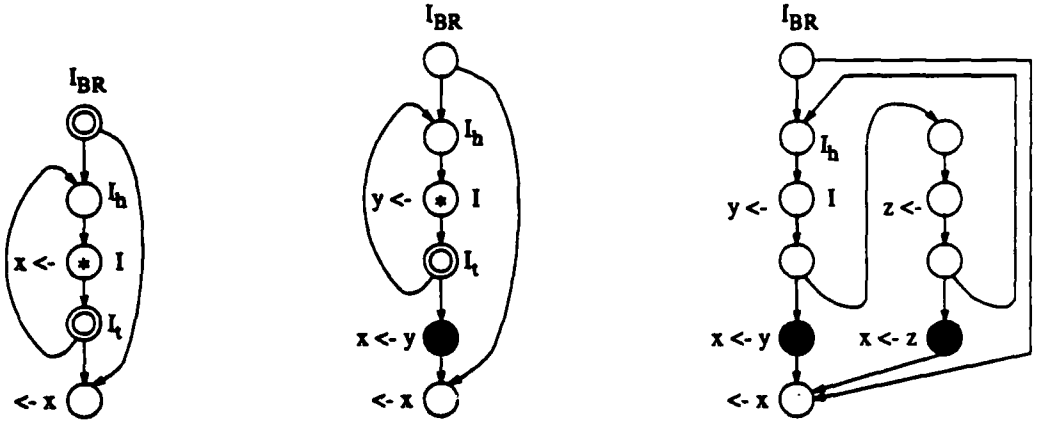
Subroutine *node-splitting* performs reaching definition analysis within loops in an MBFS order. Nodes that have multiple reaching definitions and at least one of them is a hazard node are split. The nodes in inner loops are bypassed to save the execution time. An inner loop header has multiple incoming edges, but it will not be split due to the loop protection. When a node, I, is split to several copies, each new copy has a pointer (*stamp*) linking it to I. Reaching definitions from nodes that belong to the same stamp heap are considered the same reaching definition. This implements the criterion mentioned in Section 5.2. Subroutine *replicate-loop* first marks the extended loop $\tilde{L}_i$ for all hazard registers, and then applies a BFS traversal to replicate $\tilde{L}_i$. The number of copies is obtained from *get-number-of-replications* subroutine.

6.2 Incorporating Branch Hazards – Schemes 2 and 3

Branch hazards occur at branch boundaries when an error results in a wrong branch decision. The following criterion can be used to locate all branch hazards : Register x is a branch hazard if there exists a branch node I_{BR}, such that the distance from I_{BR} to a definition of x along one branch path of I_{BR} is within N, and x is *live* at the other branch paths of I_{BR}. Similar to the case shown in Figure 5(b), we need to modify the loop protection criterion. As shown in Figure 5(c), I_α is a branch node that does not use register r, and r is live along one branch path of I_α. Loop L_i is protected from outside for register r, as if branch node I_α uses register r.

By viewing x as if it is used at I_{BR}, renaming can resolve branch hazards as well as on-path hazards. We use the following example to illustrate the idea.

Example Consider the partial segment of EQN, as shown in Figure 6(a), and $N = 4$. Register x at node I is a branch hazard due to branch nodes I_{BR} and I_t. After loop protection as in Figure 5(c), and renaming x to y, the new register y at node I is a branch hazard due to branch



(a) EQN segment, N = 4. (b) After loop protection on x. (c) After expanding twice and renaming.

Figure 6: The loop expansion for branch hazards

node I_t, as shown in Figure 6(b). Note that the save instruction $y \leftarrow x$ before the loop header I_h is removed since x is not live at I_h. In Figure 6(c), by expanding the loop twice and renaming, the branch hazard is resolved. The formula for the number of loop replications can also be modified by viewing the branch node as using the hazard register x.

7 Post-Pass Code Rescheduling and Spill Register Reassignment

7.1 On-Path Hazards – Scheme 0

Although the pseudo register phase aims at removing on-path hazards within a function, new hazards may emerge after machine register phase. First, the stack pointer adjustment instructions within the prologue segment and the epilogue segment create immediate anti-dependencies of length 1. Second, before calling a procedure, the registers used as parameters need to be saved before the new values can be loaded. Register spilling may also create on-path hazards. When a register is to be spilled, most likely it will be loaded with new values, thus creating a use-before-definition scenario. A straightforward post-pass nop insertion algorithm was employed in Scheme L to resolve these new hazards. Sufficient nops are inserted before the hazard definitions to force all anti-dependency distances exceeding N.

In this section, we apply a code rescheduling technique within the prologue and the epilogue segments, and a register reassignment algorithm for rearranging spill registers, so that the total

number of nops inserted is greatly reduced. The post-pass algorithm includes the following steps : 1) reassign spill registers; 2) reschedule code and insert nops in the prologue segment; 3) reschedule code and insert nops in the epilogue segment; and 4) insert remaining nops.

IMPACT C compiler reserves three registers as spill registers, i.e., \$3, \$24, and \$25. The spill registers perform two functions to access memory, *load* and *store*. The compiler generates instructions of the following groups for load and store functions respectively, where $\$r_1$ and $\$r_2$ are different spill registers, and are dead after the second (or the third) instruction :

load $\$r_1$, memory;	load $\$r_1$, $memory_1$;	operation defining $\$r_1$;
use $\$r_1$;	load $\$r_2$, $memory_2$;	store $\$r_1$, memory;
	use $\$r_1$, $\$r_2$;	

Spill registers are served as temporaries and have very short live ranges, i.e., 2 or 3. On-path hazards occur when two groups of spill code use the same spill register and their distance, from the use of the first group to the definition of the second group, is less than or equal to N. All groups of spill code can be easily identified. The goal is to minimize the number of nops needed to resolve all hazards. Our approach is to utilize dead registers as substitutes within groups so that the sum of all the anti-dependency distances for spill registers and substitutes is maximized, considering the anti-dependency distance between groups of different spill registers and substitutes $N + 1$. In general, this problem is NP-hard, which includes as a special case the following NP-complete problem after fixing that only spill registers are dead registers, and $N = 1$:

> *Given K colors, an undirected graph G and an integer n, is there a node coloring such that the number of edges with the same colors at both ends is at most n?*

This can be proven by restricting n to 0, and it becomes the K-colorability problem [16]. However, we propose a simple heuristic algorithm to reassign spill registers within groups in a BFS traversal of the entire program. We always choose as a substitute the register which is dead before and after the group, and whose sum of the distance backward to the first use and the distance forward to the first definition is maximum.

The prologue segment includes code to adjust the stack pointer and to save the values of some local registers to memory, while the epilogue segment includes code to retrieve the original values of the same local registers from memory and to adjust the stack pointer. We illustrate the improvement to the epilogue segment by an example, while the prologue segment can be similarly

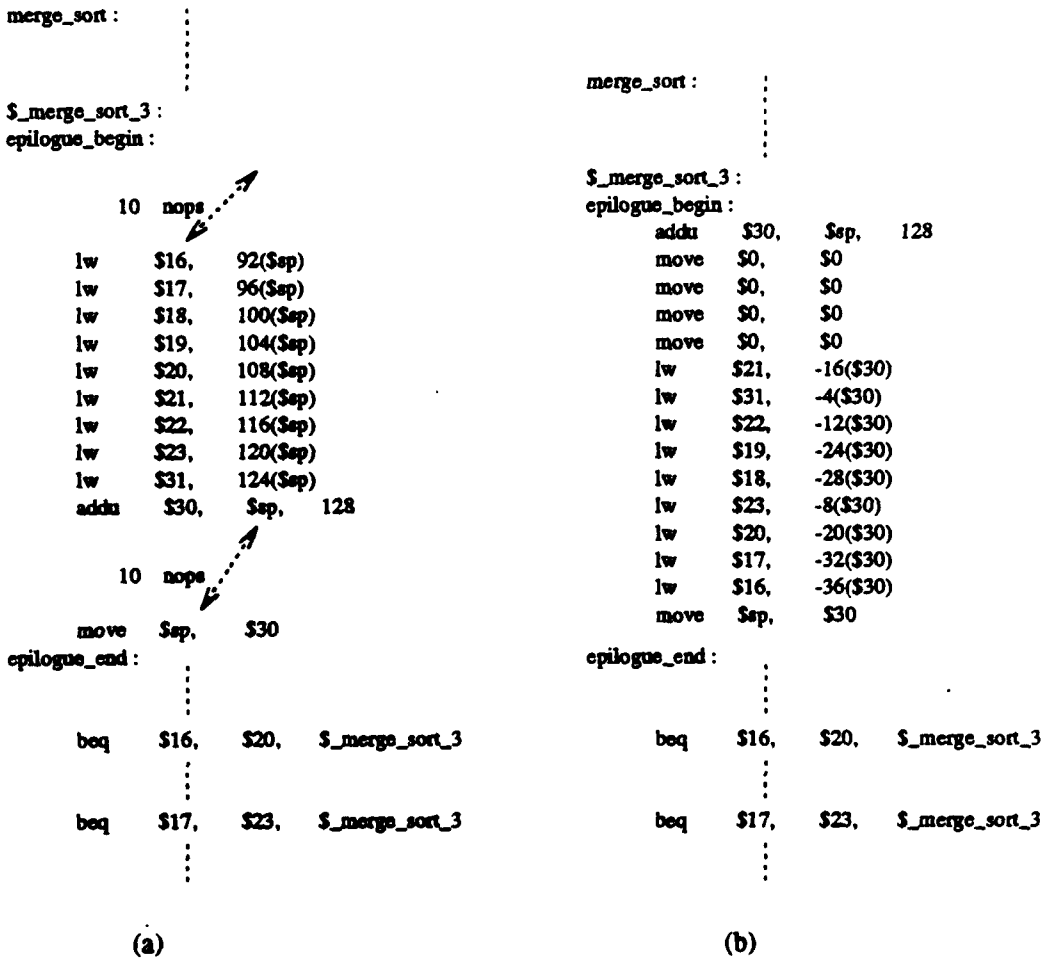


Figure 7: Post-pass code rescheduling for the epilogue segment of QSORT, $N = 10$

done. Figure 7(a) shows the epilogue segment of the second function, *merge-sort*, of QSORT, for $N = 10$. Figure 7(b) illustrates how the register assignment and code rescheduling are used to eliminate 16 nops in the epilogue segment. Instruction 'addu $30, $sp, 128' has been moved backward up to before all instructions of loading local registers, with the base register being replaced by $30. The instructions to load local registers are rescheduled according to their distances from the first uses of corresponding registers. Since registers $16, $17, $20, and $23 all have distance 1, they are moved to the end of the load instructions. Four more nops are needed to resolve the hazard register $23.

The code rearrangements within the prologue and the epilogue segments will not create on-path hazards across procedural boundaries, since we can consider a subroutine call as a single

instruction using the register that holds the return address, e.g., register \$31 in IMPACT C. The last step simply performs a BFS traversal, and inserts the required nops to resolve all remaining on-path hazards.

7.2 Both Types of Hazards – Schemes 1, 2, and 3

Post-pass nop insertion can also resolve extra branch hazards generated by the machine register allocator. The branch hazard check can be incorporated in the original on-path hazard check. The heuristic to reassign spill registers has to be modified as follows. The register we choose to replace the reserved spill register at a specific group G of spill instructions must be not only dead before and after G, but also requires as few nops as possible to resolve the new branch hazard induced by the substituting register. This can be achieved by applying an RBFS traversal from the first instruction of G, up to distance N. For every branch node I_{BR} visited, and for those registers which are live at the other branch of I_{BR}, set "the distance backward to the first use" in the heuristic to the distance from I_{BR} to G, as if those registers are used at I_{BR}. In the last step, we insert nops to resolve the remaining on-path hazards and branch hazards.

The above schemes for incorporating branch hazard resolution do not create extra hazards across procedural boundaries. However, depending on implementations, the callee-saved registers may produce a performance impact due to separate compilations.

As shown in Figure 8(a), suppose at branch node I, a wrong decision is made. After rollback and a correct decision at I, register \$r has a wrong value. If \$r is in Y's callee-saved register set, then \$r is live along I's target (T) branch. Several nops should be inserted between I and J to resolve such branch hazard. However, since Y's callee-saved register set are unknown at current procedure X, a conservative scheme may assume that all the potential registers are in the set, e.g., \$16, \$17, $\cdots$, \$23 in IMPACT C. By viewing K as a node that uses such set, we can incorporate it in the initial global live range analysis.

To relief the situation, certain remedies can be implemented. For library routines, a built-in table holding corresponding saved register sets can be attached to the compiler. The following checking can terminate \$r's live range before the procedure call, regardless of whether \$r belongs to the callee-saved register set. \$r $\in$ *live_in*(M) *iff* \$r is live at node K, where M is the next instruction following the subroutine call node K. Such live range checking starting from M should

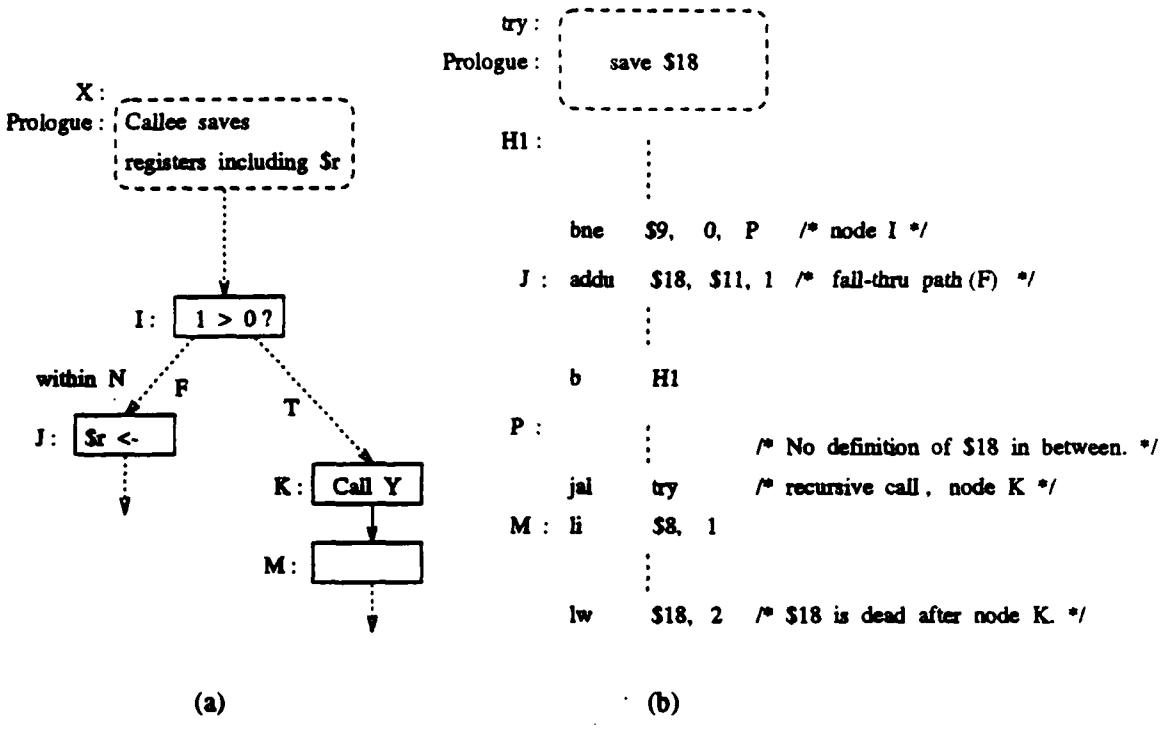


Figure 8: Register live range across procedure boundaries .

skip any subroutine call encountered.

Example Figure 8(b) is an assembly code segment for the recursive function *try* of QUEEN. Without checking the additional condition, N nops are inserted between node I and node J to eliminate the hazard $18. None is required by observing $18 is dead after node K. Code run time performance is improved since apparently such N nops are within a loop.

8 Performance Evaluation

8.1 Resolving On-Path Hazards – Scheme 0 v.s. Scheme L

The incremental updating scheme and the postpass code rescheduler improve application compile time, run-time performance, and reduce code growth for most applications studied. In this section we compare the performance impact of Scheme 0 and Scheme L with respect to the compile time, code run time and code size. For the comparison purpose, we investigate the same set of benchmarks used in [1]: CMP, COMPRESS, PUZZLE, QSORT, QUEEN, and WC.

Scheme 0 finishes compilation for all benchmarks within a short time. For $N = 10$, Scheme

Table 2: Code run time overhead

N		1	2	3	4	5	6	7	8	9	10
QSORT	L	6.2%	8.3%	8.3%	10.4%	11.5%	13.5%	14.6%	26.0%	22.9%	30.2%
	0	5.2%	6.2%	6.2%	8.3%	8.3%	10.4%	10.4%	13.5%	15.6%	16.7%
QUEEN	L	3.0%	5.3%	7.2%	7.2%	9.0%	9.8%	11.5%	15.8%	16.3%	20.9%
	0	2.9%	3.5%	3.9%	4.9%	5.1%	5.5%	6.0%	8.0%	10.2%	16.3%
CMP	L	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%
	0	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%
WC	L	3.8%	3.8%	3.8%	3.8%	3.8%	3.8%	3.8%	3.8%	3.8%	4.4%
	0	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.3%	1.3%
PUZZLE	L	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%
	0	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	0.0%	0.0%
COMPRESS	L	-0.6%	0.0%	0.0%	0.0%	1.2%	2.5%	5.6%	6.2%	11.2%	18.8%
	0	-0.6%	-0.6%	-0.6%	-0.6%	1.2%	1.2%	5.0%	5.6%	10.6%	16.9%

L spends more than 8 minutes, 15 seconds, 1.5 minutes, 3.5 minutes, and 9.5 minutes to compile benchmarks QSORT, QUEEN, CMP, WC, and PUZZLE respectively, while Scheme 0 takes compile time less than 16 seconds, 8 seconds, 15 seconds, 15 seconds and 50 seconds respectively. COMPRESS has the best compile time improvement. Scheme L spends more than an hour for $N = 7$, 8, and 9, and almost two hours for $N = 10$ to compile, while Scheme 0 takes compile times all within 3 minutes.

Table 2 lists code run time overhead using both schemes respectively. Both schemes pass through pseudo register and machine register anti-dependency resolvers and the nop inserters, generating code free from anti-dependencies. Rows marked "L" and "0" include code run time overhead of Scheme L, and Scheme 0 respectively.

Let TO_i and TU_i be Scheme L code run time and Scheme 0 code run time respectively for anti-dependency distance i. The run-time enhancement factor is defined as $-\frac{TU_i - TO_i}{TO_i}$, for $i = 1, 2, \cdots, 10$, and is plotted in Figure 9. Two benchmarks, QSORT, and QUEEN, include recursive functions and have among the largest run-time enhancement factors, for $N > 5$. Post-pass code rescheduling contributes most to these benchmarks.

Table 3 lists the code size overhead using both schemes respectively. Let SO_i and SU_i be Scheme L code size and Scheme 0 code size respectively for anti-dependency distance i. From the last column, the overheads are within 250%, for $N = 10$.

The size enhancement factor is defined as $-\frac{SU_i - SO_i}{SO_i}$ for $i = 1, 2, \cdots, 10$, and is plotted in

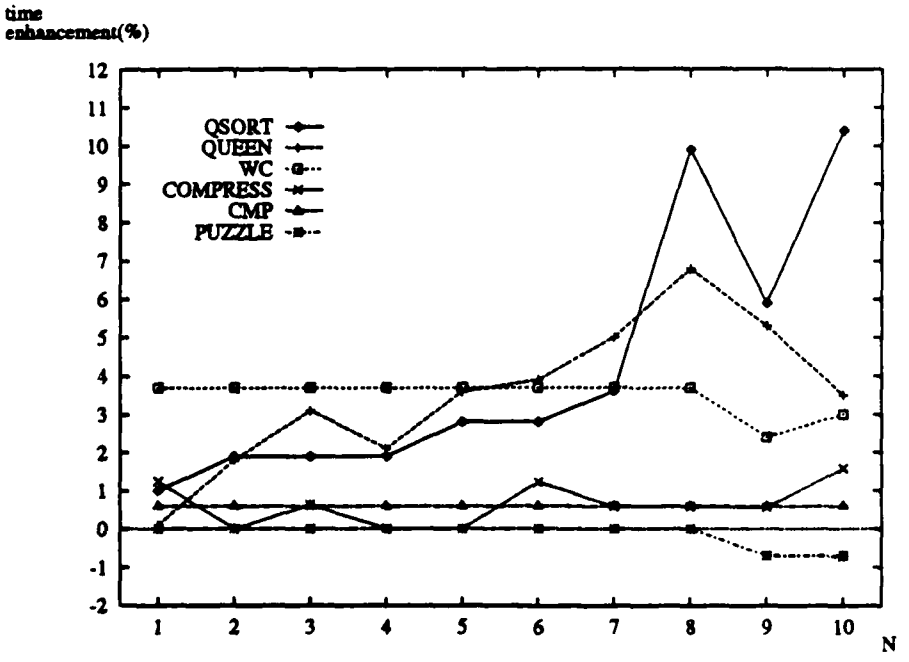


Figure 9: Run-time enhancement – Scheme 0 v.s. Scheme L

Figure 10. COMPRESS has negative size enhancement factors due to the following reasons : 1) the enhanced scheme removes the 800 instruction threshold [1] which allows further code growth; and 2) one function enters simplified mode for $N > 0$, and there are two when $N \geq 5$. For $N = 1$ and 2, QSORT and WC have negative size enhancement factors. This is because proper renaming after protecting loop L from inside and node splitting for small N may prevent loop L from being expanded, while using cut register set technique to move save/restore nodes out of the loop L requires L to be expanded at least once.

8.2 Resolving On-Path and Branch Hazards – Schemes 1, 2, and 3

Schemes 1, 2, and 3 deal with removing both types of hazards during three separate phases. Scheme 1 has the fastest compilation speed since it postpones the branch hazard resolution to the last phase, i.e., nop insertion.

All three schemes perform relatively the same for the twelve benchmarks studied. The reasons may be 1) the occurrences of branch hazards are less frequent; 2) both machine register and nop insertion phases employ heuristics, and the spill register reassignment heuristic may be efficient

Table 3: Code size overhead

N		1	2	3	4	5	6	7	8	9	10
QSORT	L	62.5%	69.7%	104.6%	114.6%	123.4%	136.0%	154.4%	199.2%	218.8%	273.9%
	0	101.1%	103.8%	105.0%	109.6%	118.0%	130.3%	138.3%	146.4%	168.6%	190.8%
QUEEN	L	56.8%	68.9%	124.3%	133.8%	152.0%	164.2%	176.4%	208.1%	218.9%	309.5%
	0	48.0%	53.4%	58.1%	68.2%	78.4%	127.0%	132.4%	147.3%	151.4%	179.1%
CMP	L	74.9%	79.7%	92.0%	106.8%	120.3%	140.6%	158.2%	179.3%	199.6%	227.5%
	0	60.2%	63.3%	66.9%	76.1%	81.7%	83.7%	87.6%	90.4%	94.0%	121.5%
WC	L	132.6%	138.1%	159.7%	166.9%	179.0%	215.5%	244.8%	248.7%	256.9%	289.5%
	0	152.5%	155.2%	160.2%	162.6%	164.1%	165.2%	187.3%	205.0%	208.8%	244.2%
PUZZLE	L	79.8%	80.3%	86.5%	89.4%	90.8%	93.7%	96.4%	101.1%	105.9%	126.0%
	0	78.7%	78.9%	80.5%	84.0%	84.5%	86.7%	95.6%	99.1%	100.5%	111.2%
COMPRESS	L	27.7%	31.6%	37.5%	52.4%	60.1%	69.0%	80.0%	93.9%	106.5%	129.0%
	0	69.7%	73.1%	74.3%	77.5%	82.0%	86.6%	107.8%	122.4%	151.8%	156.1%

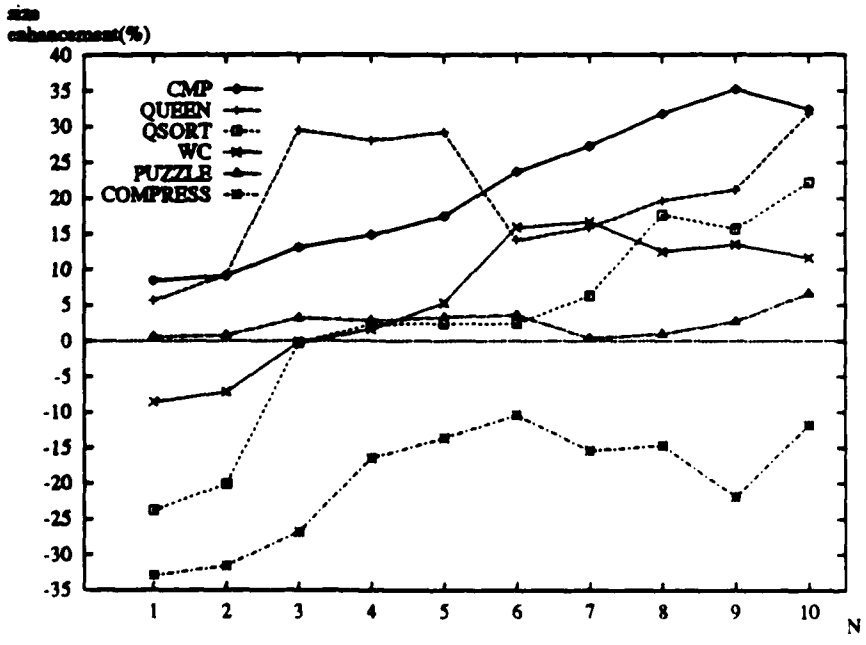


Figure 10: Size enhancement – Scheme 0 v.s. Scheme L

Table 4: Run time overhead for Scheme 1

N	1	2	3	4	5	6	7	8	9	10
QSORT	6.2%	6.2%	7.3%	9.4%	9.4%	12.5%	12.5%	16.7%	18.7%	18.7%
QUEEN	2.8%	3.1%	4.1%	5.7%	6.3%	6.7%	7.4%	11.1%	11.2%	18.0%
CMP	-3.0%	-3.0%	-3.0%	-3.0%	-3.0%	-3.0%	-2.4%	-1.8%	-1.2%	-1.2%
WC	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%
PUZZLE	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.7%	0.7%	0.7%
COMPRESS	1.3%	2.0%	2.6%	4.0%	7.3%	9.3%	9.9%	11.3%	13.9%	17.9%
GREP	11.1%	11.1%	11.1%	11.1%	11.1%	13.0%	13.0%	13.0%	14.8%	24.1%
LEX	10.5%	11.6%	11.6%	11.6%	11.6%	11.6%	12.8%	14.0%	14.0%	18.6%
EQN	7.8%	11.3%	12.2%	12.2%	12.2%	12.2%	13.9%	13.9%	13.9%	13.9%
YACC	0.0%	0.0%	2.4%	2.4%	2.4%	*7.1%	*11.9%	*16.7%	*23.8%	*28.6%
CCCP	8.5%	9.3%	10.1%	11.6%	11.6%	*17.1%	*17.1%	*19.4%	*20.9%	*26.4%
TBL	5.3%	7.9%	7.9%	7.9%	7.9%	7.9%	14.5%	14.5%	14.5%	15.8%

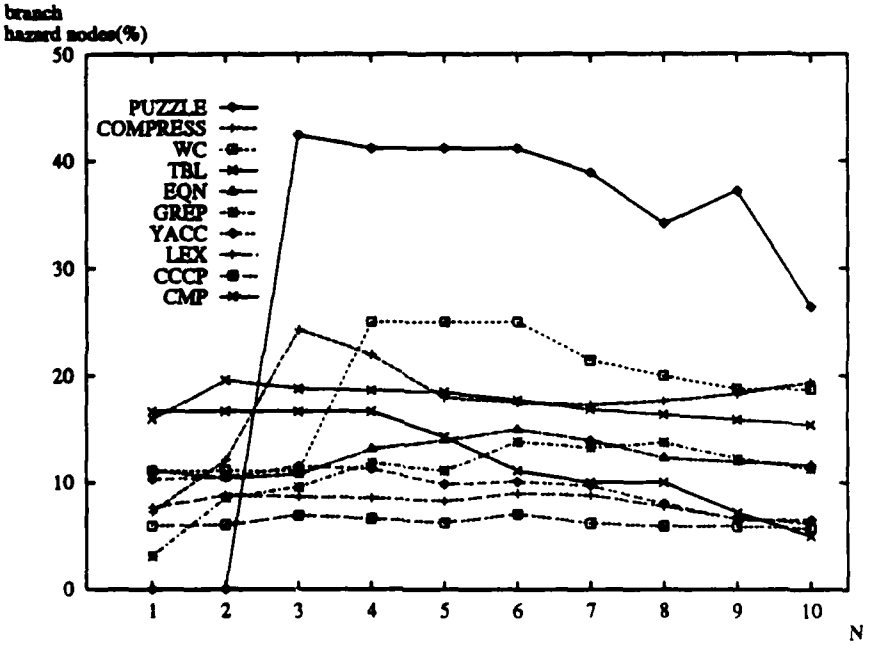


Figure 11: Percentage of hazard nodes that are branch hazard nodes

enough to resolve branch hazards in the post-pass; and 3) resolving branch hazards at the pseudo register phase or the machine register phase is likely to have larger code growth, due to the extra node splitting and loop expansion. In most benchmarks, Scheme 1 even outperforms the other two schemes in both code run-time and code growth, e.g., QUEEN, QSORT, CMP, COMPRESS, PUZZLE, and WC.

The performance overhead of Scheme 1 is tabulated in Table 4. Due to the heuristic algorithm employed in the post-pass phase, the performance overhead we observed is not monotonically increasing according to N. The code generated to allow N instruction rollback certainly can work for $N-1$ instruction rollback scheme. Therefore, we can record the overhead non-decreasingly. All twelve benchmarks successfully pass the pseudo register phase in a short time. However, there are several functions generating more than 15,000 nodes, which increases the computation time for the machine register assignment phase, when $N > 6$. YACC has two such functions, and CCCP has one. For these three functions, we resolve the rollback hazards of distance 5 in the pseudo register phase, and then resolve the rollback hazards of distance $N > 5$ in the post-pass phase, as marked by "*" in Figure 4.

Figure 11 depicts the percentage of hazard nodes that are branch hazard nodes but are not on-path hazard nodes, for various rollback distance N. Benchmarks QUEEN and QSORT have 0 percentage for N within 10 because either they have no branch hazards, or all of their branch hazards are also on-path hazards. PUZZLE has the highest percentage of branch hazard nodes, 42.42% when $N = 3$. There is a sheer rise from $N = 2$ to $N = 3$ due to the relative distances between branch nodes and hazard nodes. This can explain why in Scheme A, PUZZLE has the highest run-time overhead 10% when $N = 10$ [11]. The post-pass algorithms apparently trim down the overhead to 0.7%, as shown in Table 4. All the other benchmarks have less than a quarter of hazard nodes that are branch hazard nodes but not on-path hazard nodes.

9 Conclusion

An incremental updating scheme has been incorporated in the compiler-assisted multiple instruction retry scheme, resulting in significantly reduced compile times. To improve the code run time and to reduce the code size, several approaches have been applied. By identifying the cut register set,

save/restore nodes can be moved out of the loops during loop protection. The code in the prologue and the epilogue segments can be rescheduled, and the spill registers can be reassigned to reduce the total number of nops inserted. The threshold for the number of nodes increases from 800 to 15,000. Branch hazards can also be resolved by simple modifications to the proposed approaches. Based on the types of hazards resolved at the three different phases, we have implemented three schemes to transform the programs into code with rollback capability. Among them, Scheme 1 postpones the resolution of branch hazards to the last phase, and hence has the fastest compilation speed. It also typically generates code as good as the other two schemes in both code run time and code growth.

References

[1] C.-C. J. Li, S.-K. Chen, W. K. Fuchs, and W.-M. W. Hwu, "Compiler-assisted multiple instruction retry," Tech. Rep. CRHC-91-31, Coordinated Science Laboratory, University of Illinois, May 1991.

[2] L. Svobodova, "Resilient distributed computing," *IEEE Transactions on Software Engineering*, vol. SE-10, No. 3, May 1984.

[3] L. Lin and M. Ahamad, "Checkpointing and rollback-recovery in distributed object based systems," in *The Twentieth International Symposium on Fault-Tolerant Computing*, pp. 97–104, 1990.

[4] K. Tsuruoka, A. Kaneko, and Y. Nishihara, "Dynamic recovery schemes for distributed processes," in *IEEE 2nd Symp. on Reliability in Distributed Software and Database Systems*, pp. 124–130, 1981.

[5] C.-C. J. Li and W. K. Fuchs, "CATCH - Compiler-Assisted Techniques for CHeckpointing," in *The Twentieth International Symposium on Fault-Tolerant Computing*, pp. 74–81, June 1990.

[6] W.-M. W. Hwu and Y. N. Patt, "Checkpoint repair for high-performance out-of-order execution machines," *IEEE Transactions on Computers*, vol. C-36, pp. 1496 -1514, Dec. 1987.

[7] M. L. Ciacelli, "Fault Handling on the IBM 4341 Processor," in *The Eleventh International Symposium on Fault-Tolerant Computing*, pp. 9–12, June 1981.

[8] M. S. Pittler, D. M. Powers, and D. L. Schnabel, "System development and technology aspects of the IBM 3081 processor complex," *IBM Journal of Research and Development*, vol. 26, pp. 2-11, Jan. 1982.

[9] Y. Tamir and M. Tremblay, "High-performance fault-tolerant vlsi systems using micro rollback," *IEEE Transactions on Computers*, vol. 39, pp. 548–554, Apr. 1990.

[10] Y. Tamir, M. Liang, T. Lai, and M. Tremblay, "The UCLA Mirror Processor: a building block for self-checking self-repairing computing nodes," in *The Twenty-First International Symposium on Fault-Tolerant Computing*, pp. 178–185, June 1991.

[11] N. J. Alewine, S.-K. Chen, C.-C. J. Li, W. K. Fuchs, and W.-M. W. Hwu, "Branch recovery with compiler-assisted multiple instruction retry," in *The Twenty-Second International Symposium on Fault-Tolerant Computing*, pp. 66–73, July 1992.

[12] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[13] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. The MIT Press, 1990.

[14] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, no. 1, pp. 47–57, 1981.

[15] D. A. Padua and M. J. Wolfe, "Advanced computer optimizations for supercomputers," *Communications of the ACM*, vol. 29, pp. 1184–1201, Dec. 1986.

[16] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*. Freeman, 1979.